**Homework 3**

**Akshay Mishra**  **011476673**

**Q1. For *each* job:**

**At high level, describe the logic of your MapReduce job**

**• Input directory on the VM =**

**• Output directory on the VM =**

**• # of map tasks =**

**• # of reduce tasks =**

**Your screenshots (specified later) must match this info.**

**Answer:**

**Mapper**: This class takes care of running mapper jobs where every line of input in all input files is split based on space character, which is in-line with the structure of the provided files. The mapper retrieves appropriate element, which belongs to the URL field as per the log format. For every URL, the mapper associates count of 1, and passes this to reducer as its input.

**Reducer**: Reducer basically sums up the count for every URL and stores it in the form of URL against its count. As an additional responsibility through its cleanup method, the reducer also sorts the results based on URL count using Java collections APIs. The sorting functionality could have been achieved using another MapReduce job, but choosing overriding cleanup function over another MapReduce job is merely a design choice.

• Input directory on the VM = (inside HDFS) /tmp/wordcount/in

• Output directory on the VM =(inside HDFS)  /tmp/wordcount/out

• # of map tasks = 3

• # of reduce tasks = 1

**Q2. Use the aforementioned example hit count and sorted output to illustrate step by step *how* your jobs perform the work correctly.**

**Answer:**

Below is the Mapper class:

public class MapperAkshayMishra673 extends Mapper<Object, Text, Text, IntWritable> {

  private final static IntWritable ONE = new IntWritable(1);

  private Text word = new Text();

  public void map(Object key, Text value, Context context

                  ) throws IOException, InterruptedException {

      String[] split = value.toString().split(" ");

      for(int i=0; i<split.length; i++){

          if (i==6)

              word.set(split[i]);

              context.write(word, ONE);

      }

  }

}

* Mapper class will take input from the log files provided as per the job configuration. As per job configuration, it’ll read file from /tmp/wordcount/in dir in HDFS.
* The Mapper will split every line based on space character as below.
* For example,

198.104.162.38 - - [01/Jul/1995:23:59:49 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786 🡺

[“198.104.162.38” “-“ “-“ “[01/Jul/1995:23:59:49 -0400]” "GET” “/images/NASA-logosmall.gif” “HTTP/1.0" “200” “786”]

* The mapper will associate count of 1 with every input and will simply ignore rest of the fields. So, it will throw output in the form of key, value such that key is URL in the line and value is count ‘1’. The input format is Object & Text, Output format is text, IntWritable.
* IntWritable is a class defined in Hadoop Library.

public class ReducerAkshayMishra673 extends Reducer<Text,IntWritable,Text,IntWritable> {

    //private IntWritable result = new IntWritable();

    TreeMap<Text,IntWritable> result = new TreeMap<Text, IntWritable>();

    public void reduce(Text key, Iterable<IntWritable> values,  Context context) throws IOException, InterruptedException {

            int sum = 0;

            for (IntWritable val : values) {

                sum += val.get();

            }

            result.put(new Text(key),new IntWritable(sum));

        }

    @Override

    protected void cleanup(Context context) throws IOException, InterruptedException {

        Set<Map.Entry<Text, IntWritable>> set = result.entrySet();

        List<Map.Entry<Text, IntWritable>> list = new ArrayList<Map.Entry<Text,IntWritable>>(set);

        Collections.sort(list,

new Comparator<Map.Entry<Text, IntWritable>>() {

public int compare( Map.Entry<Text, IntWritable> o1, Map.Entry<Text,IntWritable> o2 ) {

                return ((o2.getValue()).compareTo( o1.getValue()) \* -1);

            }

        });

        for(Map.Entry<Text,IntWritable> entry:list){

            context.write(entry.getKey(),entry.getValue());

        }

    }

}

* The reducer class basically performs two functions:
  + Reducing the input provided by Mapper
  + Sorting it based on value or the URL count in the context of above example
* **Reducing**: Reduction is performed by adding the count of 1s to get a total count for every URL and then storing them inside a map. It is stored in order to further pass it to the cleanup method, wherein we have defined the logic of sorting.
* **Sorting**: The sorting logic involves storing the map.entrySet() into a set from which we are going to create a list. The list, which is a part of powerful Collection framework in Java can be sorted using Collection APIs.
* We have to define a new Comparator for this to work as Comparing Map<Map.Entry<Text, IntWritable>> is not a built-in operation in Java.
* Once sorting is done, the Reducer gives its output by writing to context, which is the final output of our reducer.
* It is written to HDFS in /tmp/wordcount/out dir as specified by the job configuration.

**Q3. Enclose screenshot s for *each* of the following steps for *each* job:**

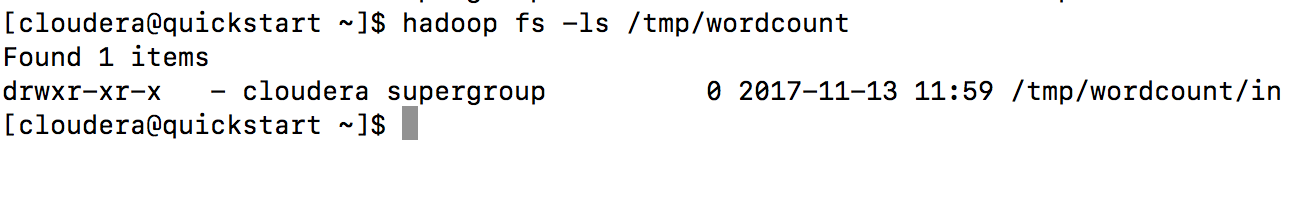
**The JAR file is stored in the /tmp directory, in order to run the program, please run below command:**

hadoop jar /tmp/hw3AkshayMishra673.jar com.cmpe282.hw3AkshayMishra673.WCAkshayMishra673 /tmp/wordcount/in /tmp/wordcount/out

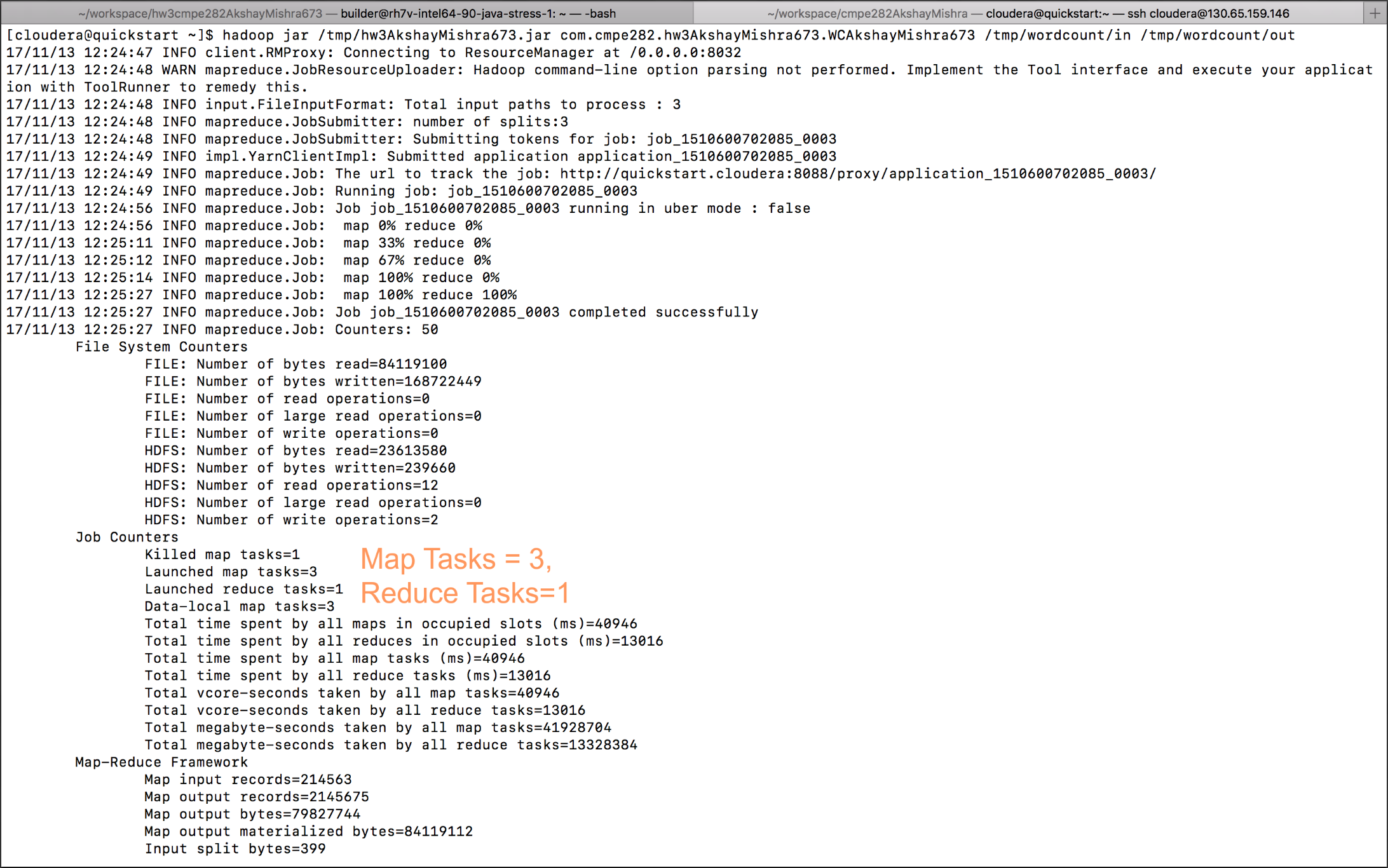
**Ensure that the /tmp/wordcount/in is present and contains all weblog files.**

**And /tmp/wordcount/out is not present, it’ll be created by the program**

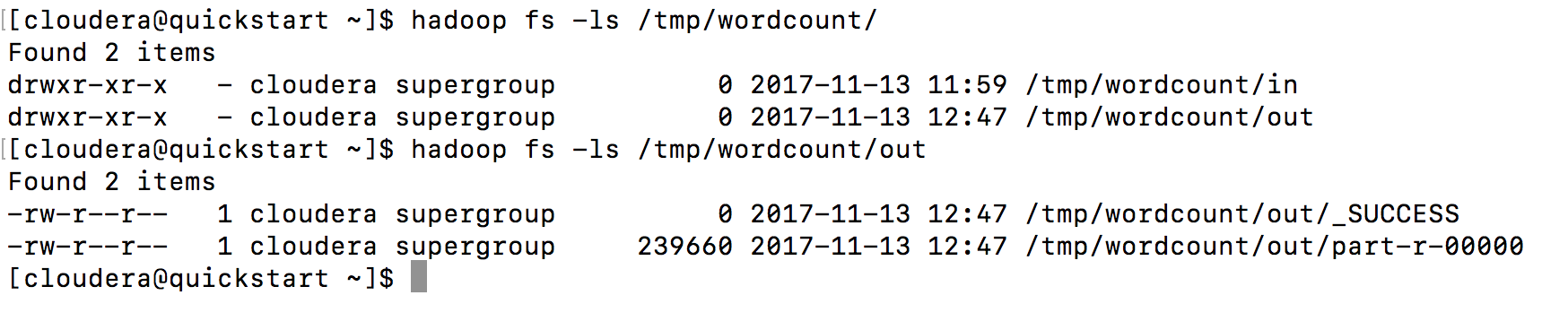
**• Before job execution, show input directory on VM o hadoop fs - ls <inputDir>**



**• During execution, capture output from “hadoop jar ... “. In particular, highlight the # of map tasks and # of reduce tasks.**

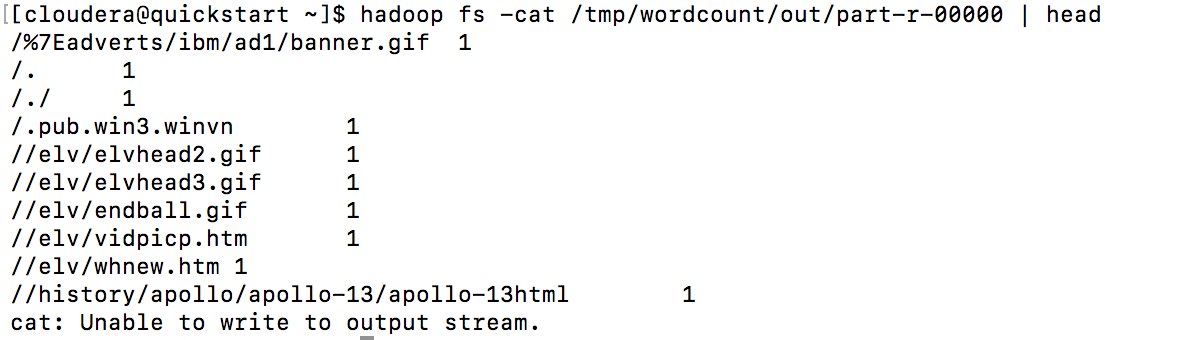


**• After job execution, show output directory on VM o hadoop fs - ls <outputDir>**

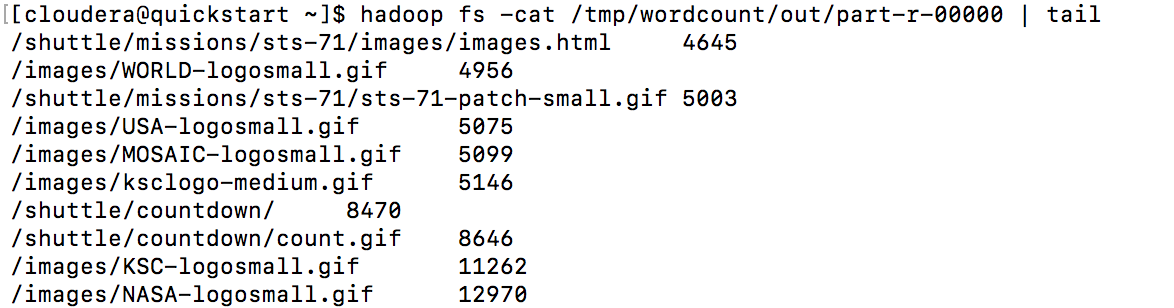


**• After execution, for *each* part - r - \* output file, show first few lines with head and last few lines with tail:**

**o hadoop fs - cat <output File> | head**



**o hadoop fs - cat <outputFile> | tail**



**Q 4. Comment on the performance and scalability of the 2nd MapReduce job. Discuss if there is any way to improve its performance and scalability.**

**Answer:**

Since I am using only one MapReduce Job, instead of second separate MapReduce Job, I’ll discuss the scalability and performance of the sorting function in my only MapReduce job.

**MapReduce Aspect:**

If we were using second MapReduce job, then it is worth noting that it is generally a better approach to keep the data in sorted state using let’s say BigTable concepts, etc. In short, it is more efficient to sort the data once during insertion that to sort them at the time of each MapReduce query. Use of combiner and more no of reducers should be able to perform data parallel reducing response time.

**Sorting Aspect:**

I am using Collections.sort() API from powerful Collection framework in Java, which is basically using list.sort() method. The current solution indirectly uses MergeSort through Collections.sort() method. MergeSort is fast, stable and guarantees O(nlogn) efficiency, although it takes extra space of o(n). We could implement our own QuickSort to sort based on the values, which would eliminate auxiliary space requirement.

Scalability of the current sorting implementation can be improved by adding more powerful machines to the hadoop cluster if we were to run the program for larger files. The faster processer with higher RAM would help to scale the application.

EXTRA CREDIT

**“wc.py”**

import sys

from pyspark import SparkContext, SparkConf

if \_\_name\_\_ == "\_\_main\_\_":

# create Spark context with Spark configuration

  conf = SparkConf().setAppName("Spark Count")

  sc = SparkContext(conf=conf)

#to count the words in a file hdfs:/// of file:/// or localfile "./samplefile.txt"

rdd=sc.textFile("hdfs://localhost:8020/tmp/wordcount/in/")#.map(lambda x: x[0]).collect()

#or you can initialize with your list

#v1='Hi hi hi bye bye bye word count'

#rdd=sc.parallelize([v1])

#print rdd.take(10)

wordcounts=rdd.map(lambda l: l.split()[6]) \

          .map(lambda w:(w,1)) \

          .reduceByKey(lambda a,b:a+b) \

          .map(lambda (a,b):(b,a)) \

          .sortByKey(ascending=True)

output = wordcounts.collect()

for (count,word) in output:

    print("%s: %i" % (word,count))

1. Save above code in current dir as “wc.py”
2. Command to run the apache spark job:

spark-submit --master yarn --deploy-mode client --executor-memory 1g --name wordcount --conf "spark.app.id=wordcount" wc.py

**Explanation:**

**Below are the sequence of actions happening in the python file.**

1. **Initialize a SparkContext**
2. **Read textfiles from specified path into the RDD**
3. **Split them and take only URL from the log lines**
4. **Associate count of 1 with every URL**
5. **Reduce based on key i.e. URL here**
6. **Swap key,value pair to value,key to facilitate sorting using SortByKey function**
7. **Collect the output**
8. **Print the output**

**Pre-requisites:**

1. Install JDK, Scala to set up the Apache Spark environment.
2. Install spark, add it to the PATH

Create below dir structure in hdfs /tmp dir.

Copy all weblog files to form a dir structure as below

/tmp

/wordcount

/in

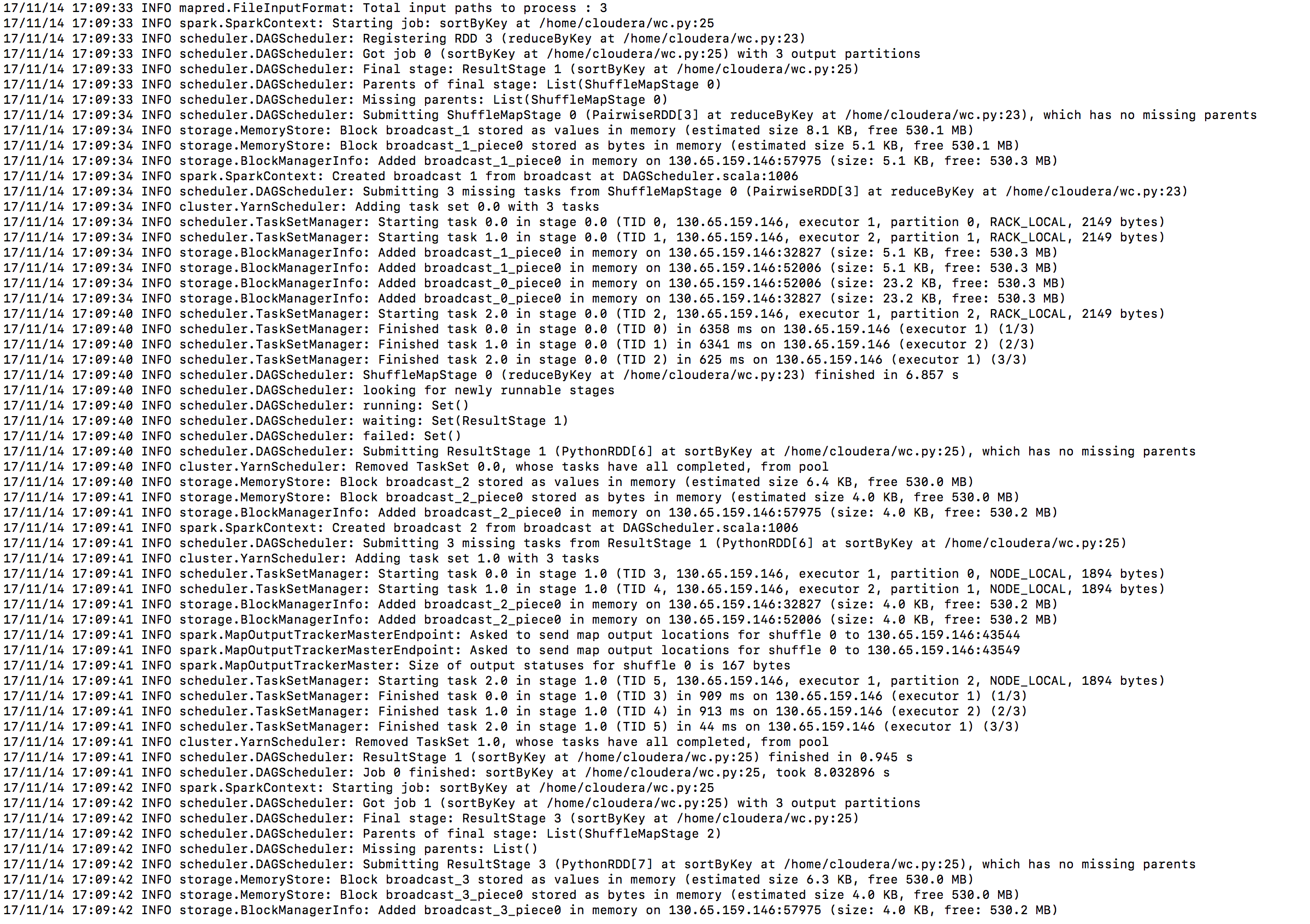
weblog-1995-7-1.txt

weblog-1995-7-2.txt

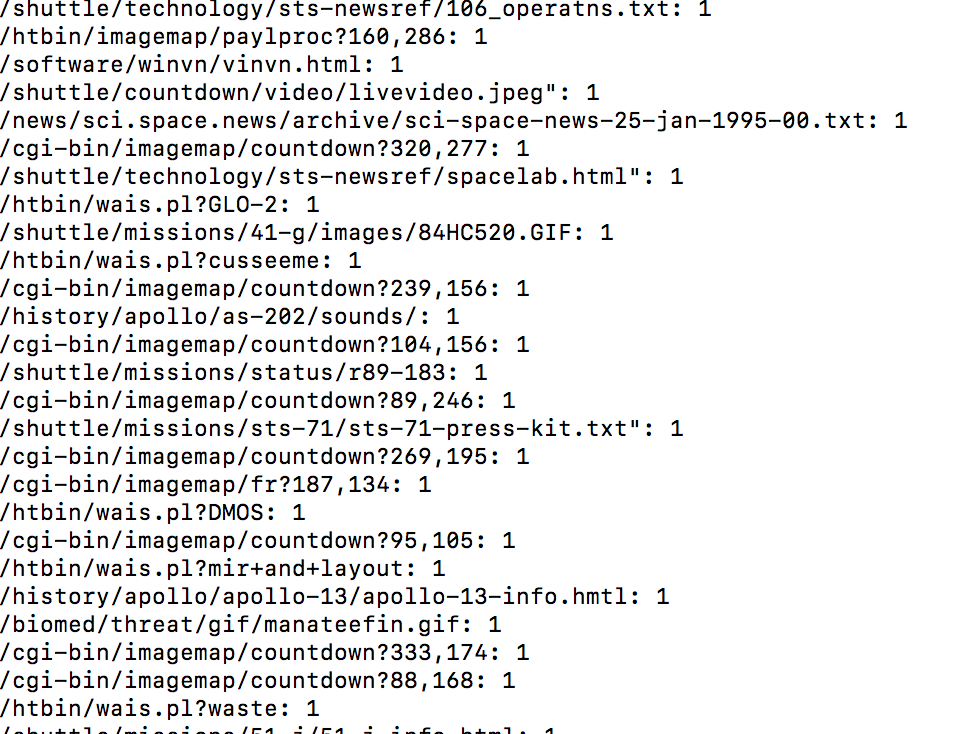
weblog-1995-7-3.txt

**Screenshots:**

Program terminal output:



Sample initial lines of output



Sample terminal lines of output:

